Info 206: Computing

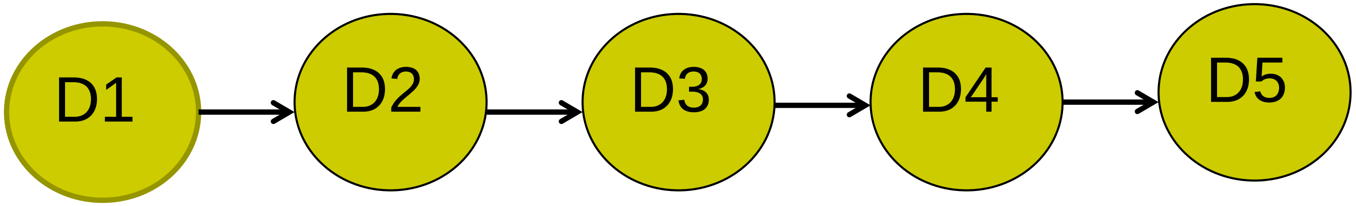
Lecture 3

Arrays, Linked Lists, and Trees

September 10, 2015

Linear Collections

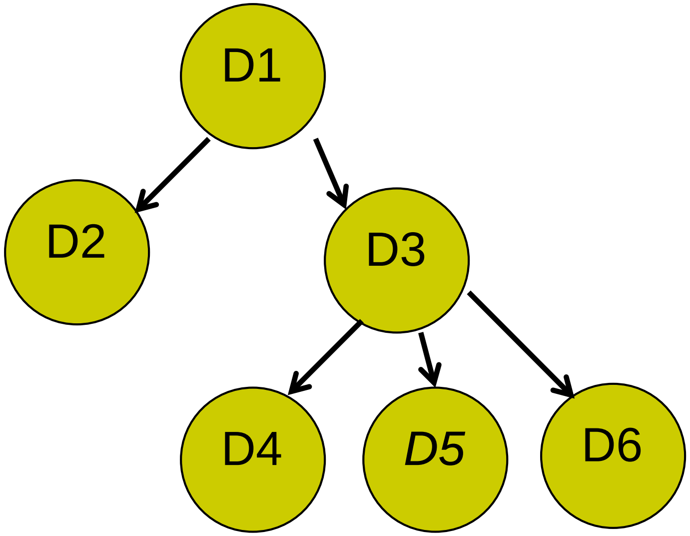
- Ordered by position



- Everyday examples:
  - Grocery lists
  - Stacks of dinner plates
  - A line of customers waiting at a bank

Hierarchical Collections

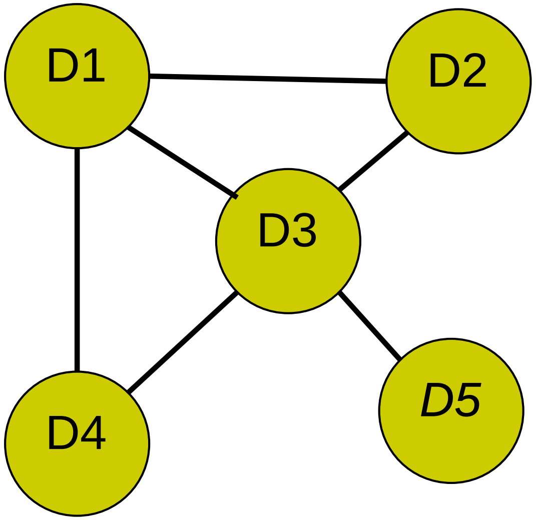
- Structure reminiscent of an upside-down tree



  - D3’s **parent** is D1; its **children** are D4, D5, and D6
- Examples: a file directory system, a company’s organizational tree, a book’s table of contents

Graph Collections

- **Graph**: Collection in which each data item can have many predecessors and many successors

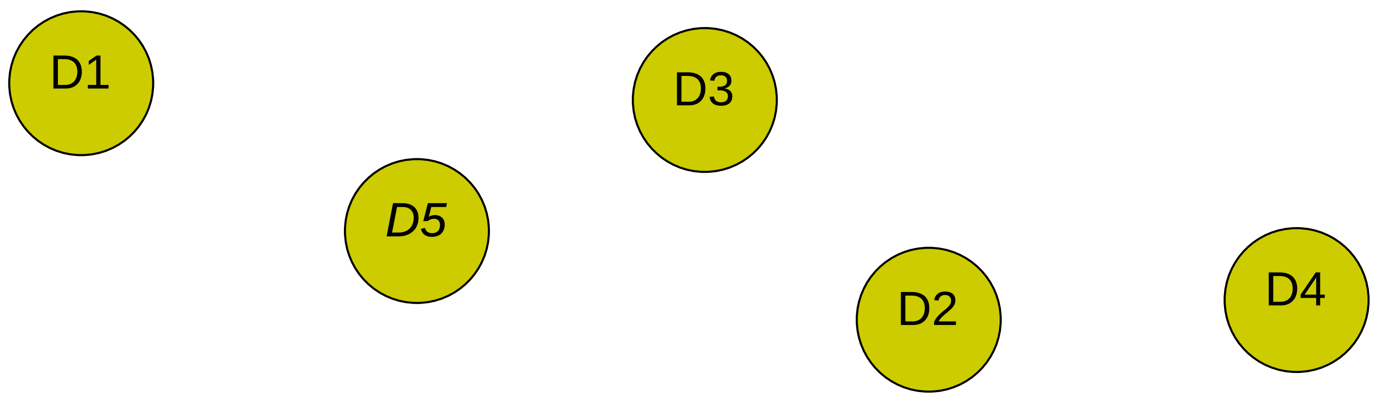


  - D3's **neighbors** are its predecessors and successors

- Examples: Maps of airline routes between cities; electrical wiring diagrams for buildings

Unordered Collections

- Items are not in any particular order
  - One cannot meaningfully speak of an item’s predecessor or successor



- Example: Bag of marbles

Operations on Collections

- Search and retrieval
- Removal
- Insertion
- Replacement (removal/insertion)
- Traversal
  - If we can traverse with Python `for` loop, then *iterable*

Operations on Collections

- Tests for equality
  - On elements in a collection
  - On the collection as a total
- Determine size
  - Some collections may have maximum capacity
- Cloning
  - Sometimes clone collections contain the same items
  - Deep copy: clone both collection and items

Abstraction and Abstract Data Types

- To a user, a collection is an abstraction
- In CS, collections are **abstract data types (ADTs)**
  - ADT users are concerned with learning its interface
  - Developers are concerned with implementing their behavior in the most efficient manner possible
- In Python, methods are the smallest unit of abstraction, classes are the next in size, and modules are the largest
- We will implement ADTs as classes or sets of related classes in modules

Data Structures for Implementing Collections: Arrays

- “Data structure” and “**concrete data type**” refer to the internal representation of an ADT’s data
- The two data structures most often used to implement collections in most programming languages are **arrays** and **linked structures**
  - Different approaches to storing and accessing data in the computer’s memory
  - Different space/time trade-offs in the algorithms that manipulate the collections

Python hides data organization

- Lists (arrays – but they can grow)
- Tuples (array on heap)
- Dictionaries (hash tables)

- But it is there “under the covers”
- Sometimes we need to more explicitly address data organization

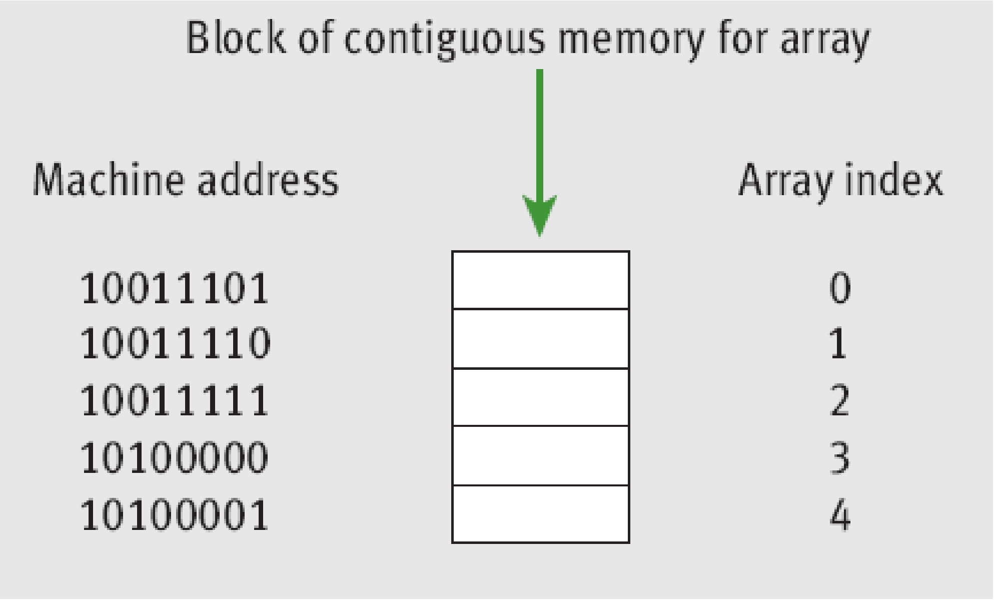
The Array Data Structure

- Array: Underlying data structure of a Python list
    - More restrictive than Python lists
- We'll define an **`Array`** class

User's Array Operation	Method in the Array Class
`a = Array(10)`	`__init__(capacity, fillValue=None)`
`len(a)`	`__len__()`
`str(a)`	`__str__()`
`for item in a:`	`__iter__()`
`a[index]`	`__getitem__(index)`
`a[index] = newitem`	`__setitem__(index, newItem)`

Random Access and Contiguous Memory

- Array indexing is a **random access** operation



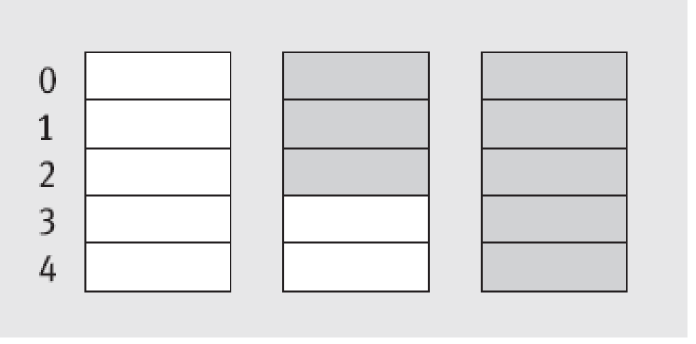
- Address of an item: **base address + offset**
  - Index operation has two steps:
    - Fetch the base address of the array’s memory block
    - Return the result of adding the index*k to this address

Static and Dynamic Arrays

- Arrays in older languages were static
- Modern languages support **dynamic arrays**
- To readjust length of an array at run time:
  - Create an array with a reasonable default size at start-up
  - When it cannot hold more data, create a new, larger array and transfer the data items from the old array
  - When the array seems to be wasting memory, decrease its length in a similar manner
- These adjustments are automatic with Python lists

Physical Size and Logical Size

- The **physical size** of an array is its total number of array cells
- The **logical size** of an array is the number of items currently in it



- To avoid reading **garbage**, must track both sizes

Physical Size and Logical Size (continued)

- In general, the logical and physical size tell us important things about the state of the array:
  - If the logical size is 0, the array is empty
  - Otherwise, at any given time, the index of the last item in the array is the logical size minus 1.
  - If the logical size equals the physical size, there is no more room for data in the array

Operations on Arrays

- We now discuss the implementation of several operations on arrays
- In our examples, we assume the following data settings:

```
DEFAULT_CAPACITY = 5
logicalSize = 0
a = Array(DEFAULT_CAPACITY)
```

- These operations would be used to define methods for collections that contain arrays

Increasing the Size of an Array

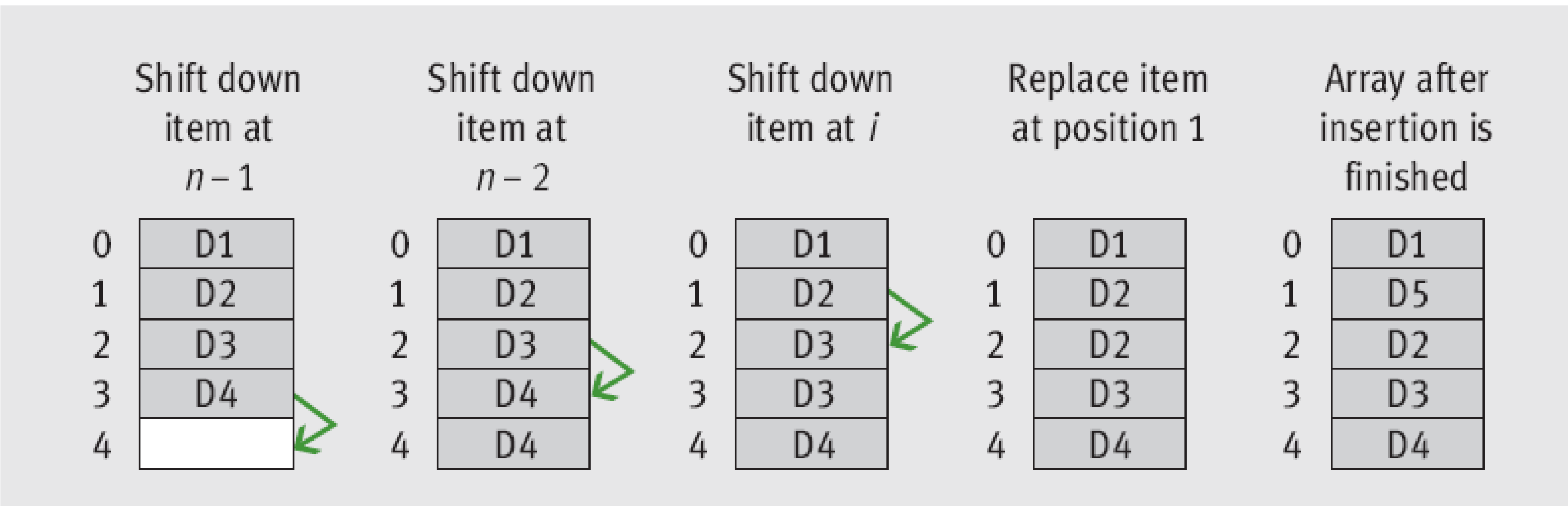
- The resizing process consists of three steps:
  - Create a new, larger array
  - Copy the data from the old array to the new array
  - Reset the old array variable to the new array object

```
if logicalSize == len(a):
    temp = Array(len(a)*2)         #Create new array
    for i in xrange(logicalSize): #Copy from old
        temp[i] = a[i]            #to new array
    a = temp # reset old array variable to new array
```

Inserting an Item into an Array

- Programmer must do four things:
  - Check for available space and increase the physical size of the array, if necessary
  - Shift items from logical end of array to target index position down by one
    - To open hole for new item at target index
  - Assign new item to target index position
  - Increment logical size by one

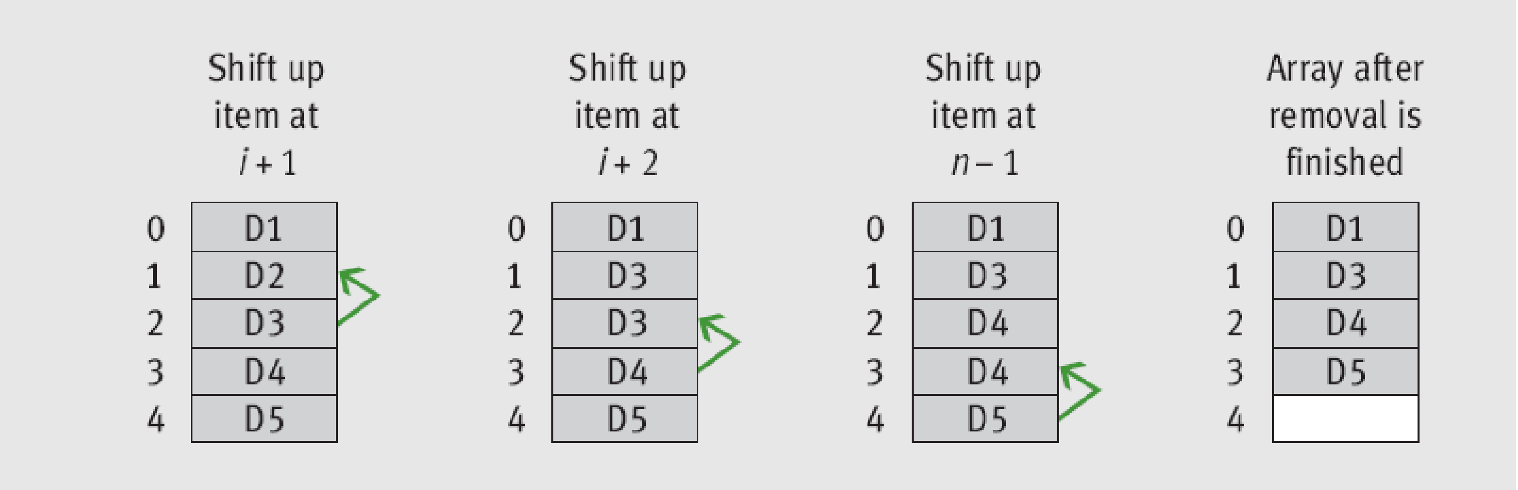
Inserting an Item into an Array



Removing an Item from an Array

- Steps:
  - Shift items from target index position to logical end of array up by one
    - To close hole left by removed item at target index
  - Decrement logical size by one
  - Check for wasted space and decrease physical size of the array, if necessary
- Time performance for shifting items is linear on average; time performance for removal is linear

Removing an Item from an Array



Complexity Trade-Off: Time, Space, and Arrays

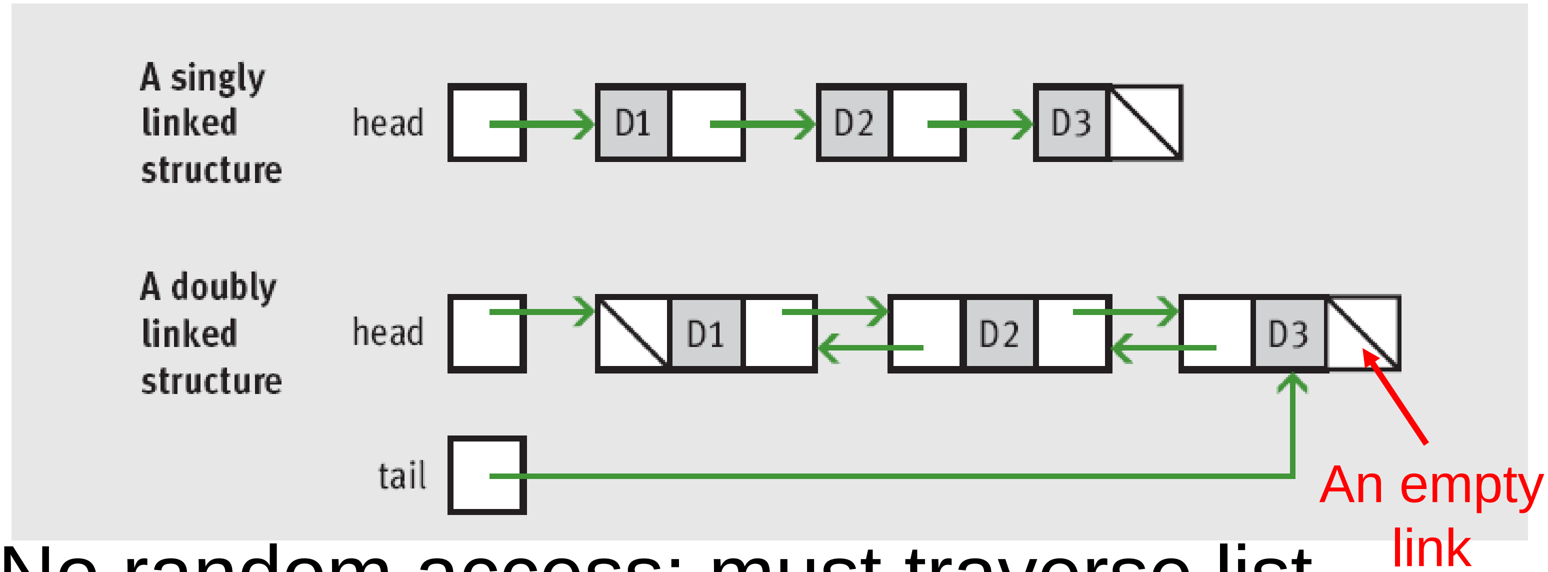
Operation	Running Time
Access at i-th position	O(1) (best & worst case)
Replacement at i-th position	O(1) (best & worst case)
Insert at logical end	O(1) (average case)
Remove from logical end	O(1) (average case)
Insert at i-th position	O(n) (average case)
Remove from i-th postion	O(n) (average case)
Increase capacity	O(n) (best & worst case)
Decrease capacity	O(n) (best & worst case)

- Average-case use of memory for is O(1)
- Memory cost of using an array is its **load factor**

Linked Structures

- After arrays, linked structures are probably the most commonly used data structures in programs
- Like an array, a linked structure is a concrete data type that is used to implement many types of collections, including lists
- We discuss in detail several characteristics that programmers must keep in mind when using linked structures to implement any type of collection

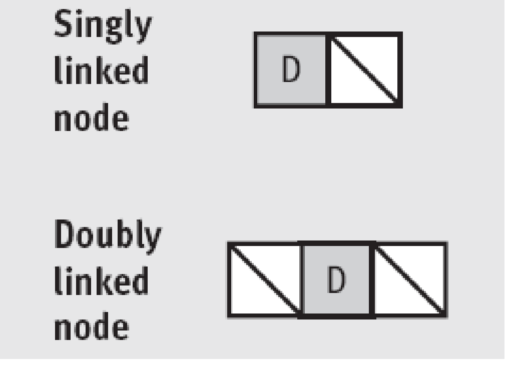
Singly Linked Structures and Doubly Linked Structures



- No random access; must traverse list
- No shifting of items needed for insertion/removal
- Resize at insertion/removal with no memory cost

Noncontiguous Memory and Nodes

- A linked structure decouples logical sequence of items in the structure from any ordering in memory
  - **Noncontiguous memory** representation scheme
- The basic unit of representation in a linked structure is a **node**:



Noncontiguous Memory and Nodes (continued)

- Ways to set up nodes to use noncontiguous memory (continued):
  - Using **pointers** (a **`null`** or **`nil`** represents the empty link as a pointer value)
    - Memory allocated from the **object heap**
  - Using **references** to objects (e.g., Python)
    - In Python, **None** can mean an empty link
    - Automatic garbage collection frees programmer from managing the object heap
- In the discussion that follows, we use the terms link, pointer, and reference interchangeably

Operations on Singly Linked Structures

- Almost all of the operations on arrays are index based
  - Indexes are an integral part of the array structure
- Emulate index-based operations on a linked structure by manipulating links within the structure
- We explore how these manipulations are performed in common operations, such as:
  - Traversals
  - Insertions
  - Removals

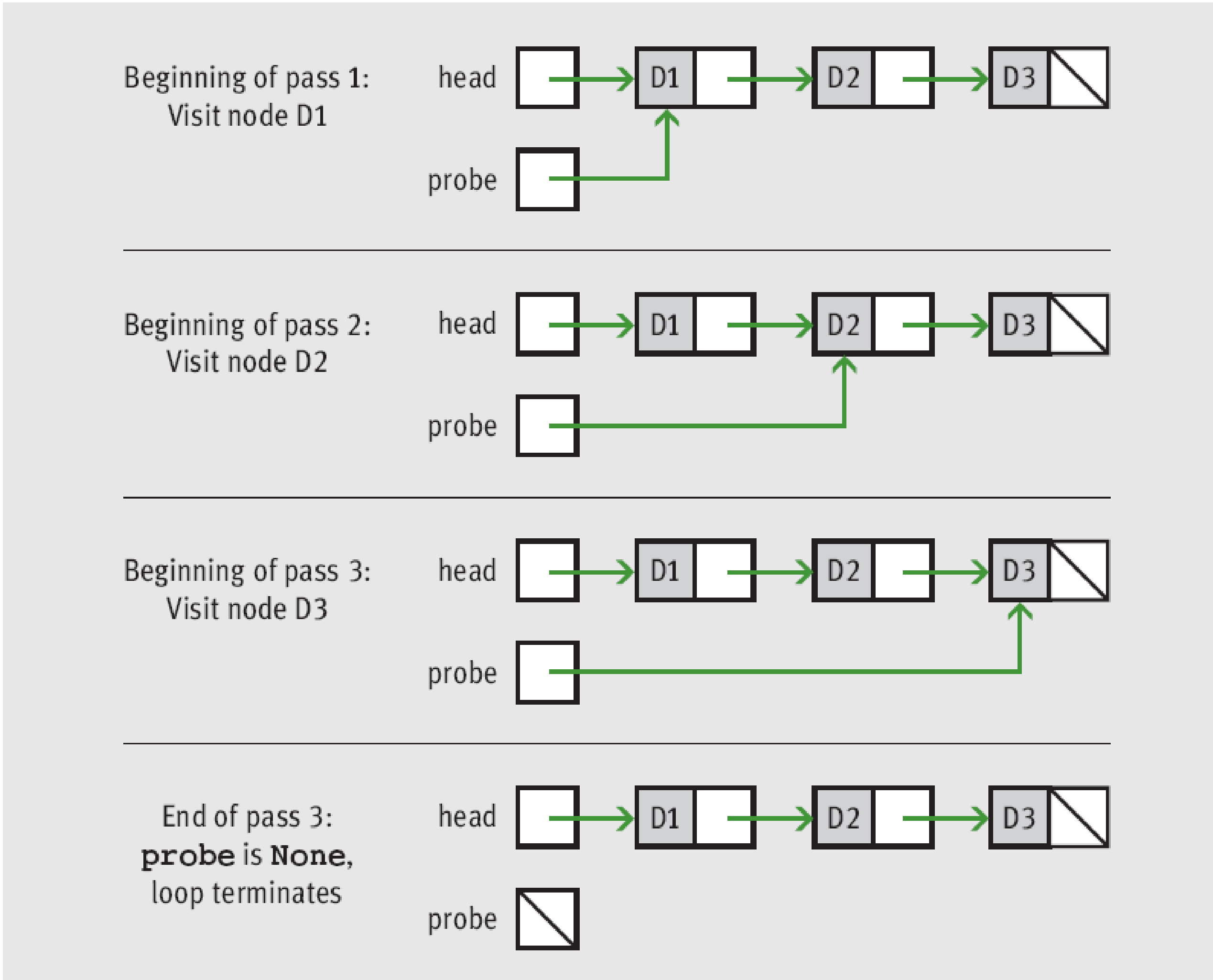
Traversal

- **Traversal**: Visit each node without deleting it
  - Uses a temporary pointer variable
- Example:

```
probe = head
while probe != None:
    <use or modify probe.data>
    probe = probe.next
```

  - **None** serves as a **sentinel** that stops the process
- Traversals are linear in time and require no extra memory

Traversal (continued)



Searching

- Resembles a traversal, but two possible sentinels:
  - Empty link
  - Data item that equals the target item
- Example:

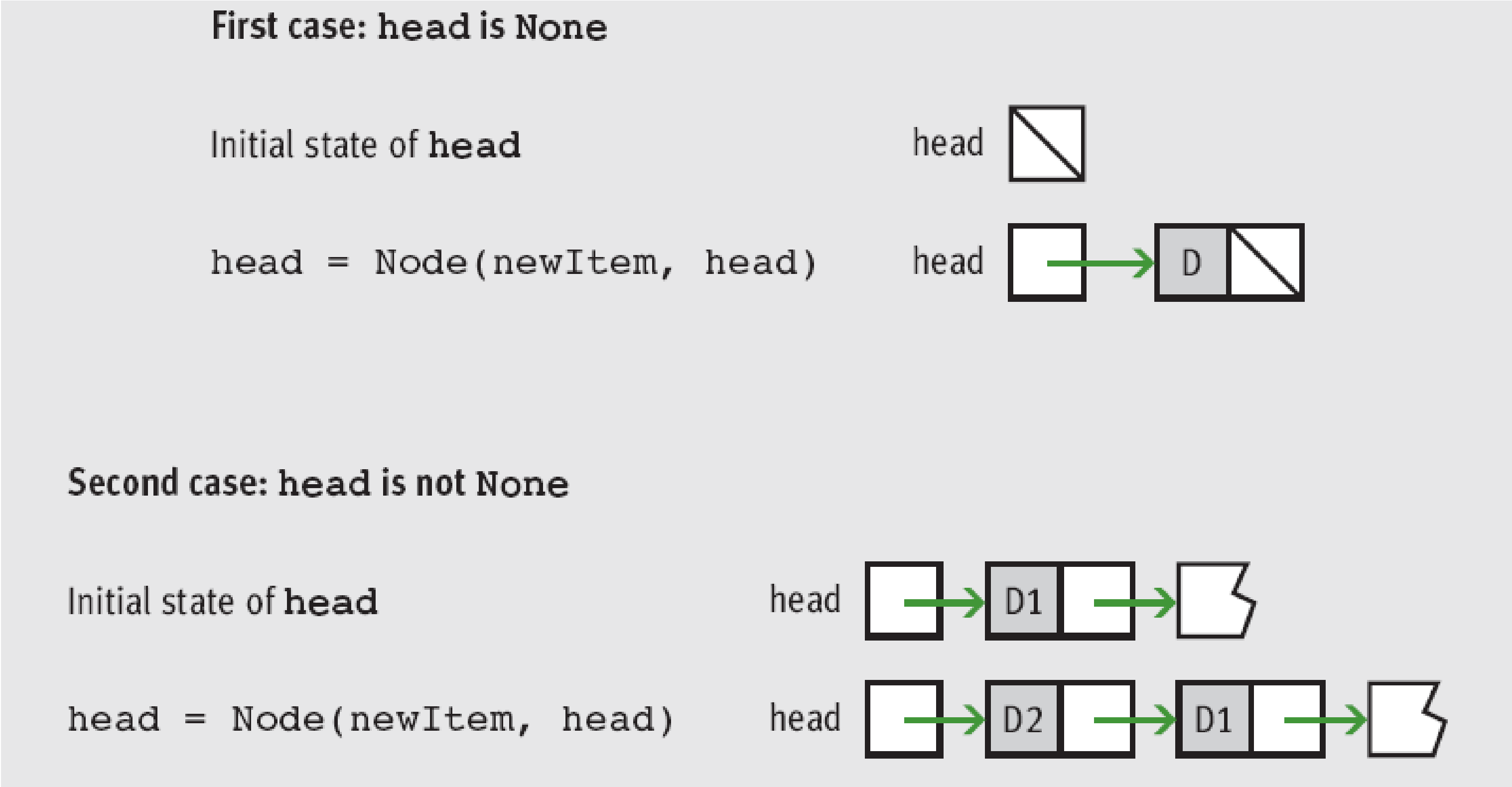
```
probe = head
while probe != None and targetItem != probe.data:
    probe = probe.next
if probe != None:
    <targetItem has been found>
else:
    <targetItem is not in the linked structure>
```

- On average, it is linear for singly linked structures

Searching (continued)

- Unfortunately, accessing the i-th item of a linked structure is also a sequential search operation
    - We start at the first node and count the number of links until the i-th node is reached
- Linked structures do not support random access
    - Can’t use a binary search on a singly linked structure
    - Solution: Use other types of linked structures

Inserting at the Beginning

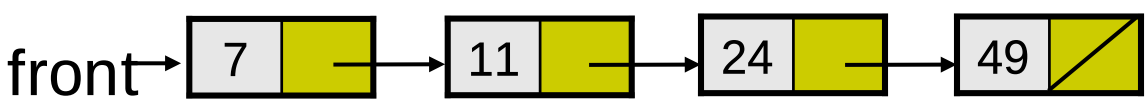


- Uses constant time and memory

Question

- Say you want to write a collection optimized for these tasks:
  - storing/accessing elements in sorted order
  - adding/removing elements in order
  - searching the collection for a given element
- What implementation would work well?
  - An array?
  - A sorted array?

index	0	1	2	3
value	7	11	24	49



  - A linked list?

Runtime

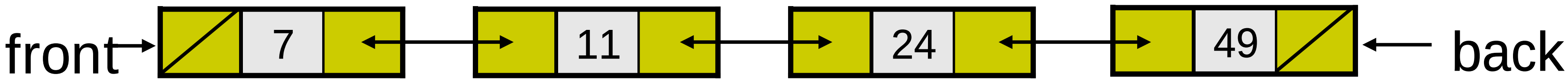
- How long does it take to do the following:
  - add N elements?
  - search for an element N times in a list of size N?
  - (add an element, then search for an element) N times?

operation	**unsorted array**	**sorted array**	**linked list**
add			
remove			
search			
access all in order			

Creative use of arrays/links

- Some data structures (such as hash tables and binary trees) are built around clever ways of using arrays and/or linked lists.
  - What array order can help us find values quickly later?

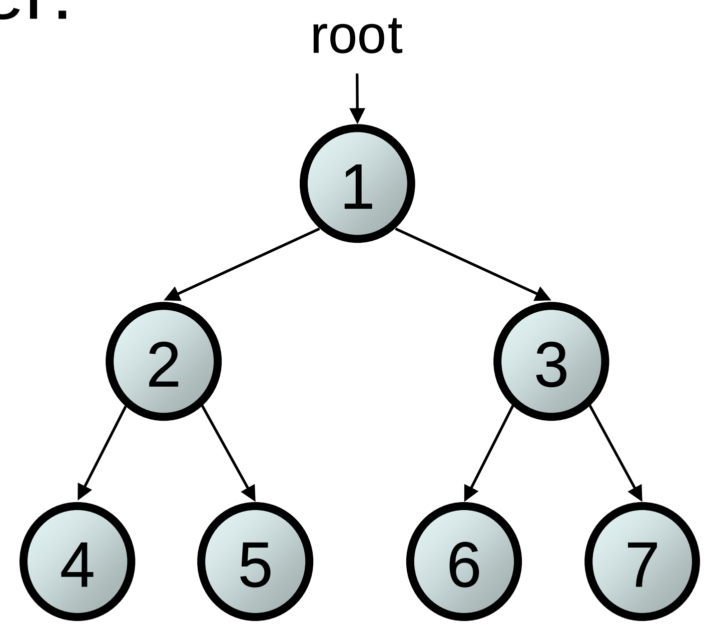
index	0	1	2	3	4	5	6	7	8	9
value	0	11	0	0	24	0	0	7	0	49



  - What if our linked list nodes each had more than one link?

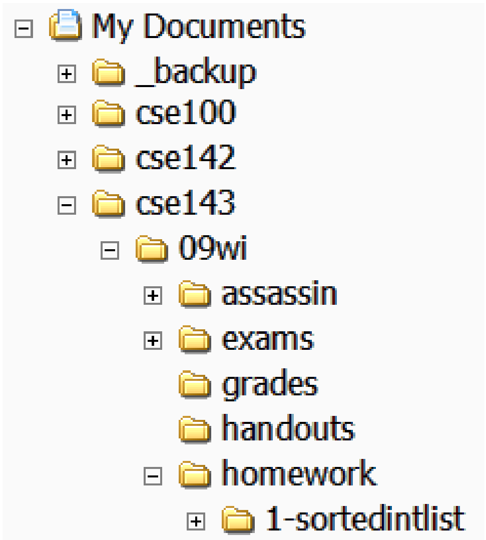
Trees

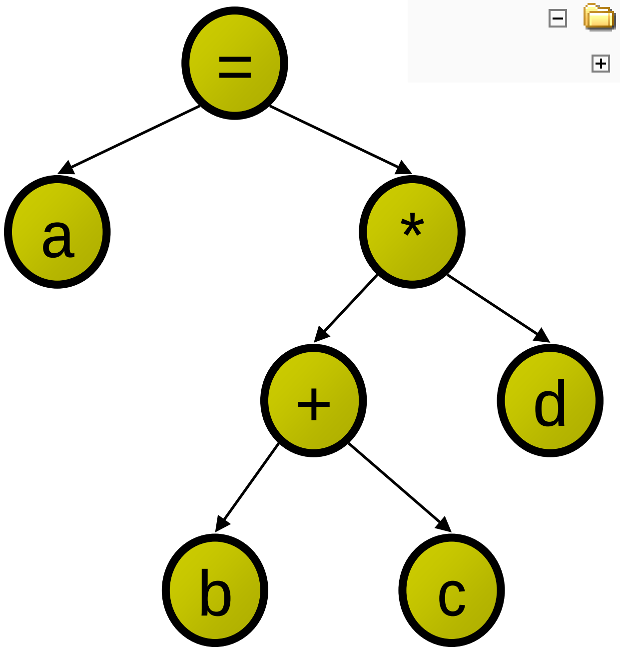
- **tree**: directed, acyclic structure of linked nodes.
  - *directed* : Has one-way links between nodes.
  - *acyclic* : No path wraps back around to the same node twice.
  - **binary tree**: Each node has at most two children.
- A tree can be defined as either:
  - empty (`null`), or
  - a **root** node that contains:
    - **data**,
    - a **left** subtree, and
    - a **right** subtree.
      - (The left and/or right subtree could be empty.)

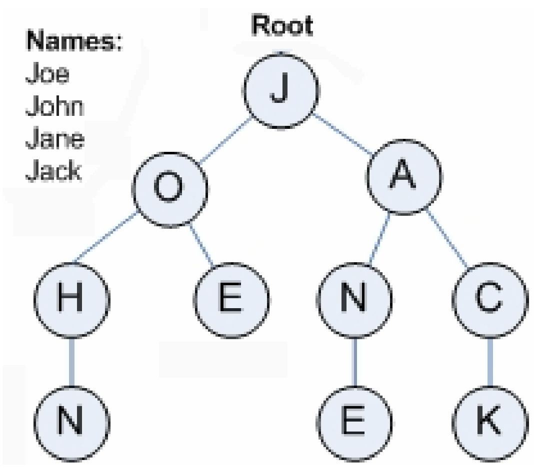


Trees in computer science

- folders/files on a computer
- family genealogy;
- organizational charts
- AI: decision trees
- compilers: parse tree
  - a = (b + c) * d;





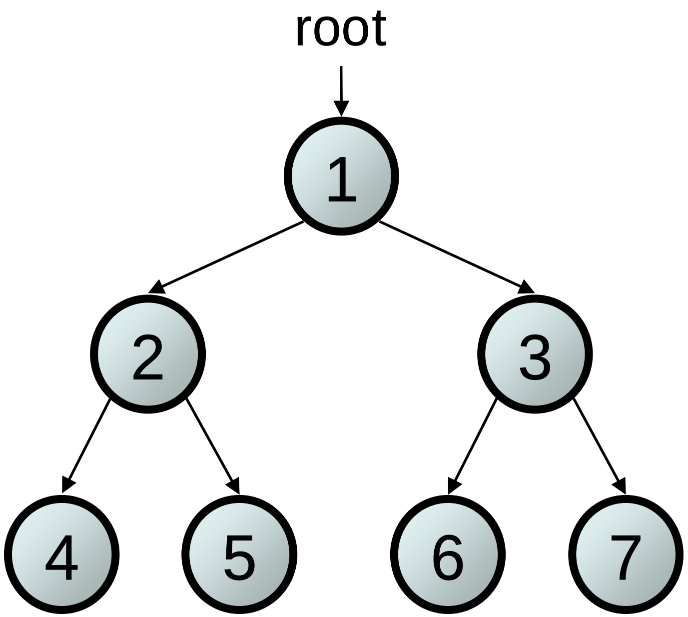


Programming with trees

- Trees are a mixture of linked lists and recursion
  - considered very elegant (perhaps beautiful!) by computer nerds
  - difficult for novices to master
- Common student comment #1:
  - "My code does not work, and I don't know why."
- Common student comment #2:
  - "My code works, and I don't know why."

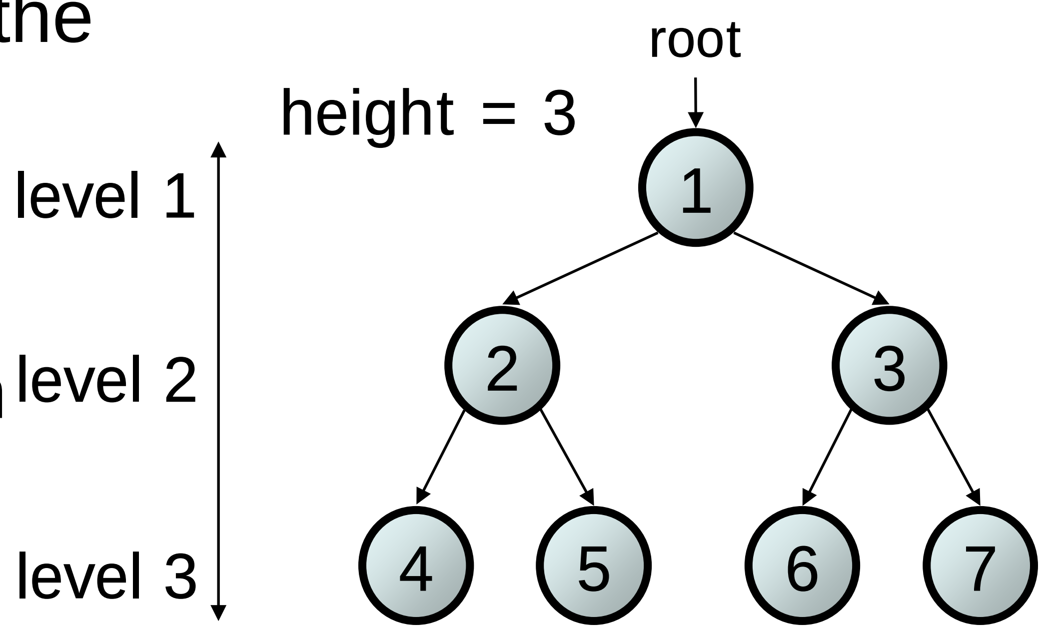
Terminology

- **node**: an object containing a data value and left/right children
- **root**: topmost node of a tree
- **leaf**: a node that has no children
- **branch**: any internal node; neither the root nor a leaf
- **parent**, **child**, **sibling**



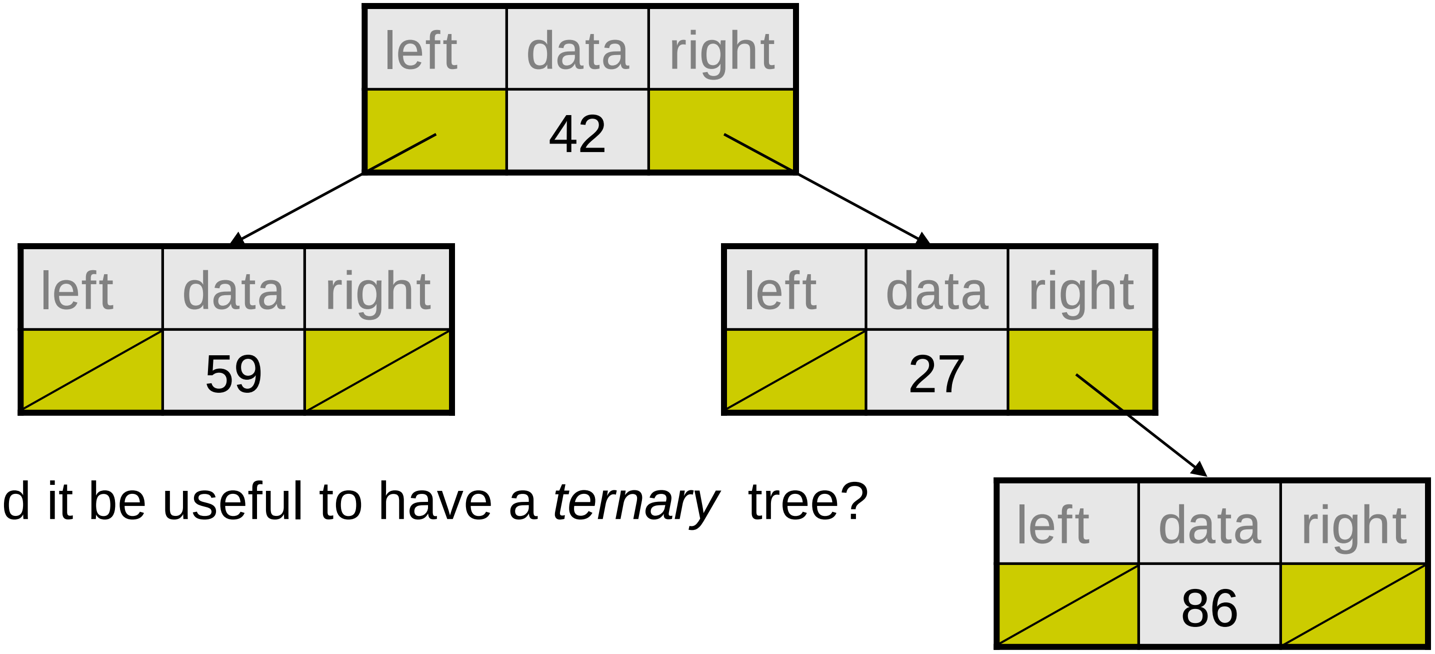
Terminology 2

- **subtree**: the tree of nodes reachable to the left/right from the current node
- **height**: length of the longest path from the root to any node
- **level**: the length of the path from a root to a given node
- **full tree**: one where every branch has 2 children



A tree node for integers

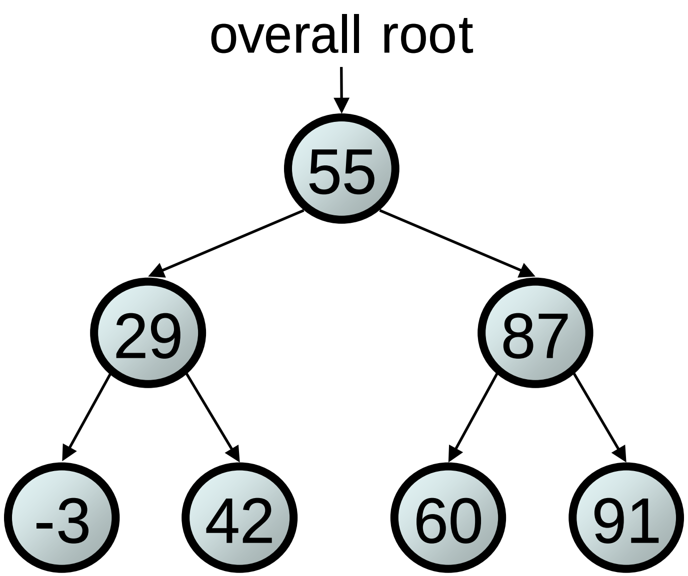
- A basic tree node object stores data and references to left/right
- Multiple nodes can be linked together into a larger tree



    - Would it be useful to have a *ternary* tree?

Binary search trees

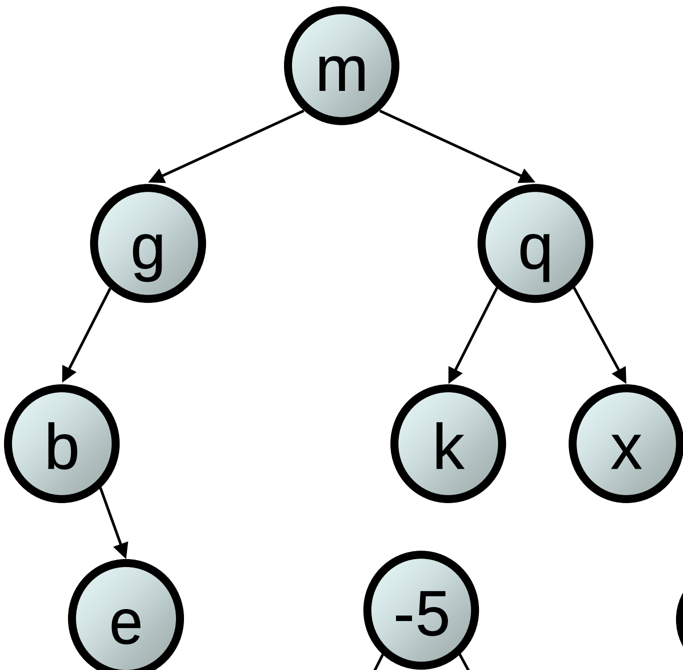
- **binary search tree** ("BST"): a binary tree that is either:
  - empty (`null`), or
  - a root node R such that:
    - every element of R's left subtree contains data "less than" R's data,
    - every element of R's right subtree contains data "greater than" R's,
    - R's left and right subtrees are also binary search trees.

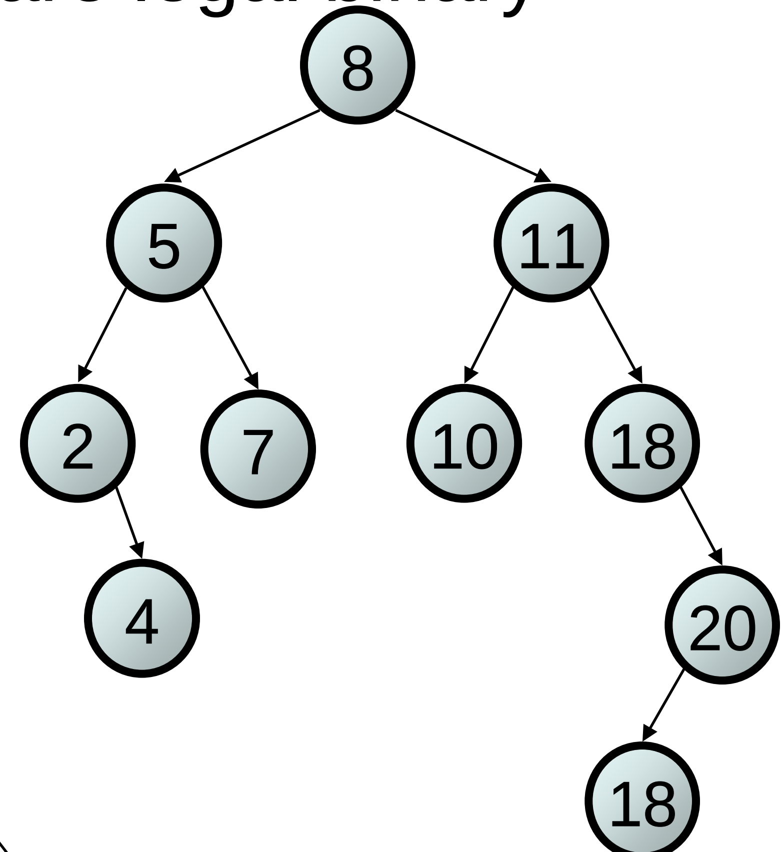


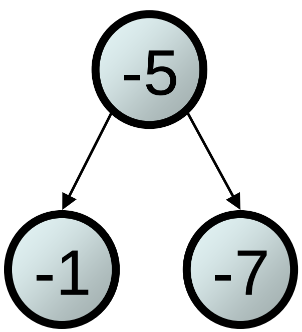
- BSTs store their elements in sorted order, which is helpful for searching/sorting tasks.

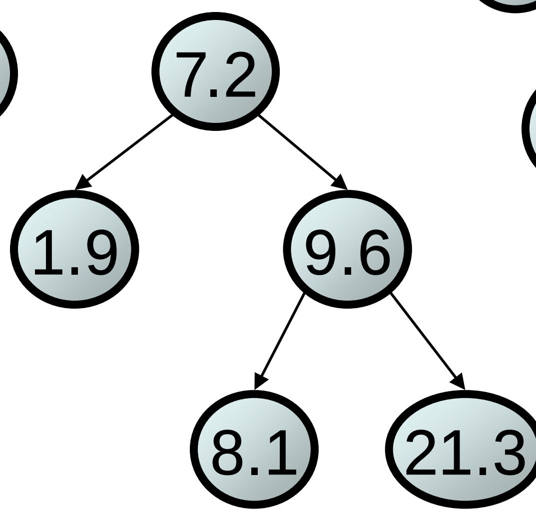
Exercise

- Which of the trees shown are legal binary search trees?



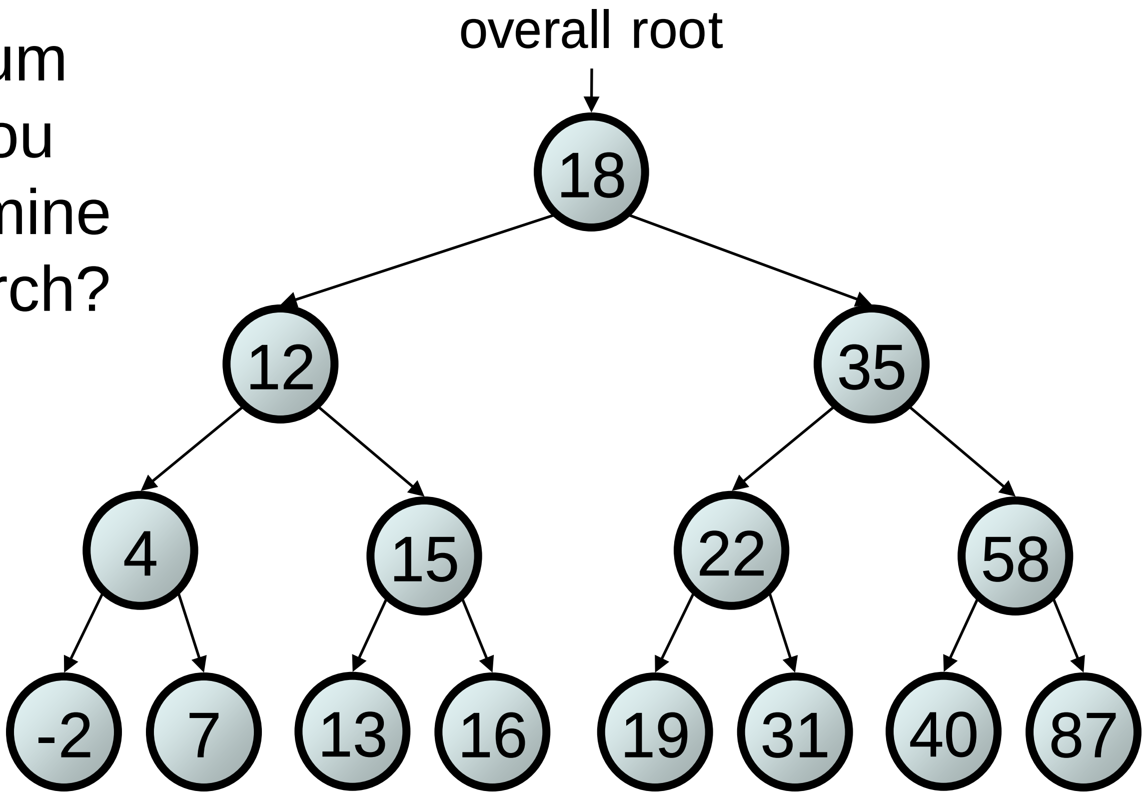






Searching a BST

- Describe an algorithm for searching the tree below for the value 31.
- Then search for the value 6.
- What is the maximum number of nodes you would need to examine to perform any search?



Adding to a BST

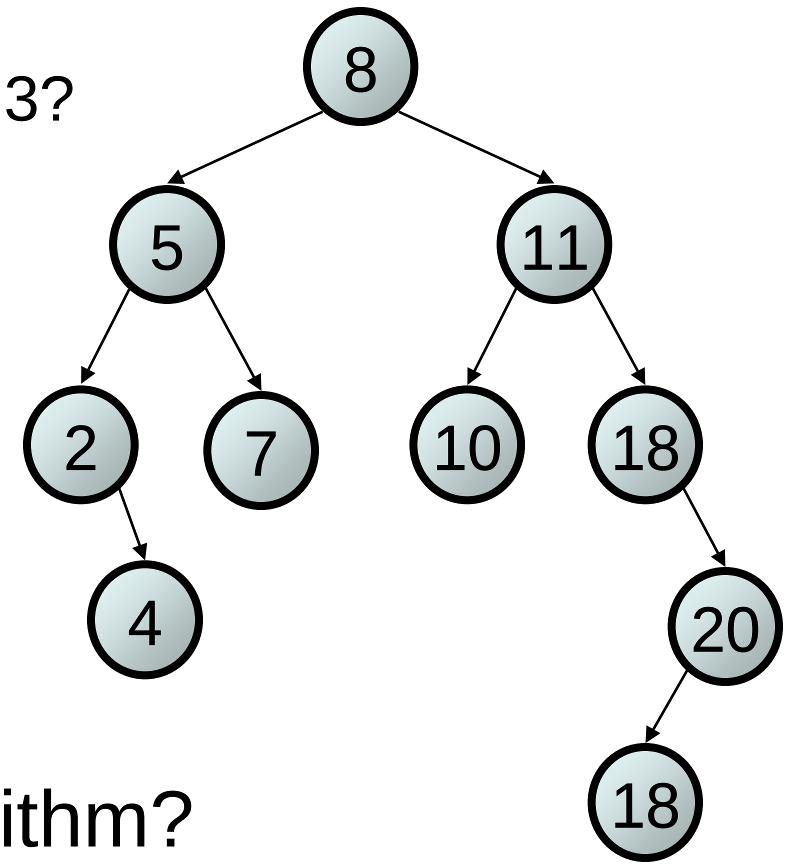
- Suppose we want to add the value 14 to the BST below.
  - Where should the new node be added?

- Where would we add the value 3?

- Where would we add 7?

- If the tree is empty, where should a new value be added?

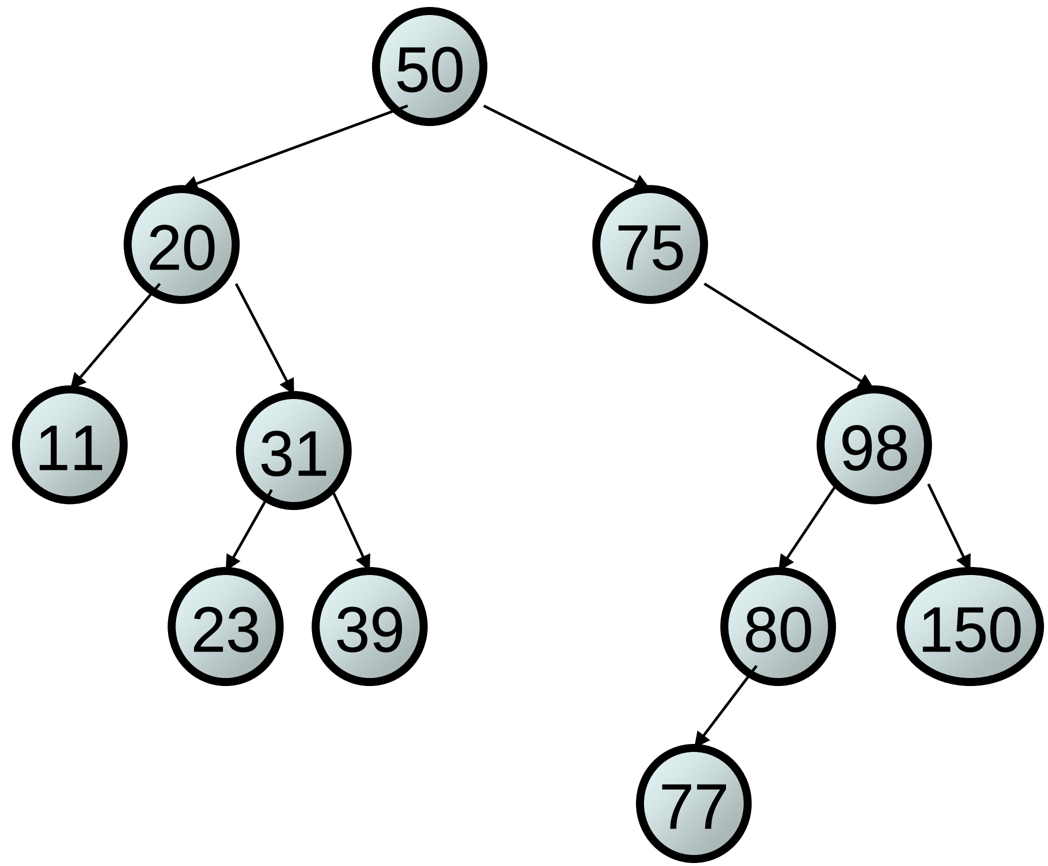
- What is the general algorithm?



Adding exercise

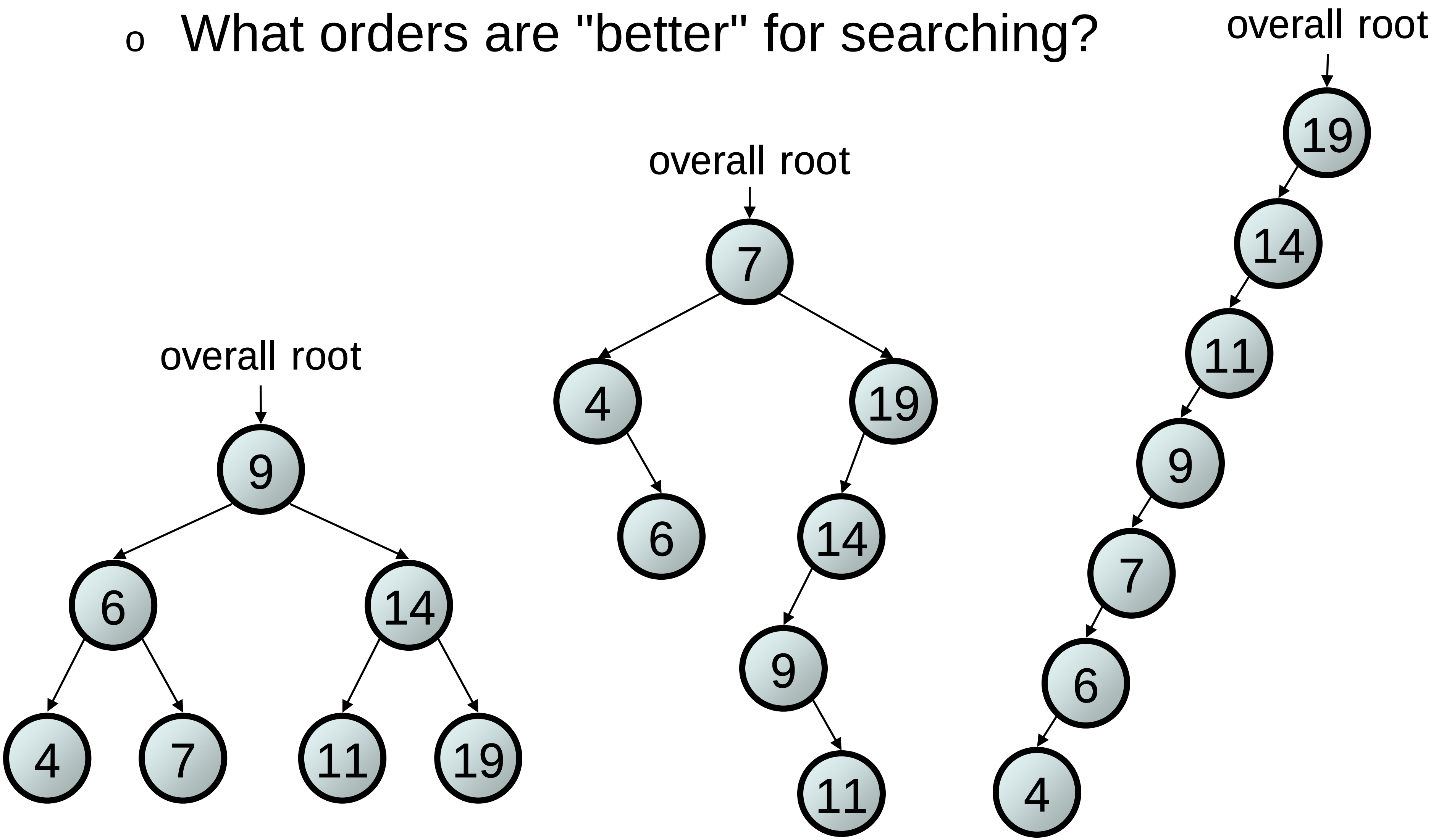
- Draw what a binary search tree would look like if the following values were added to an initially empty tree in this order:

50
20
75
98
80
31
150
39
23
11
77



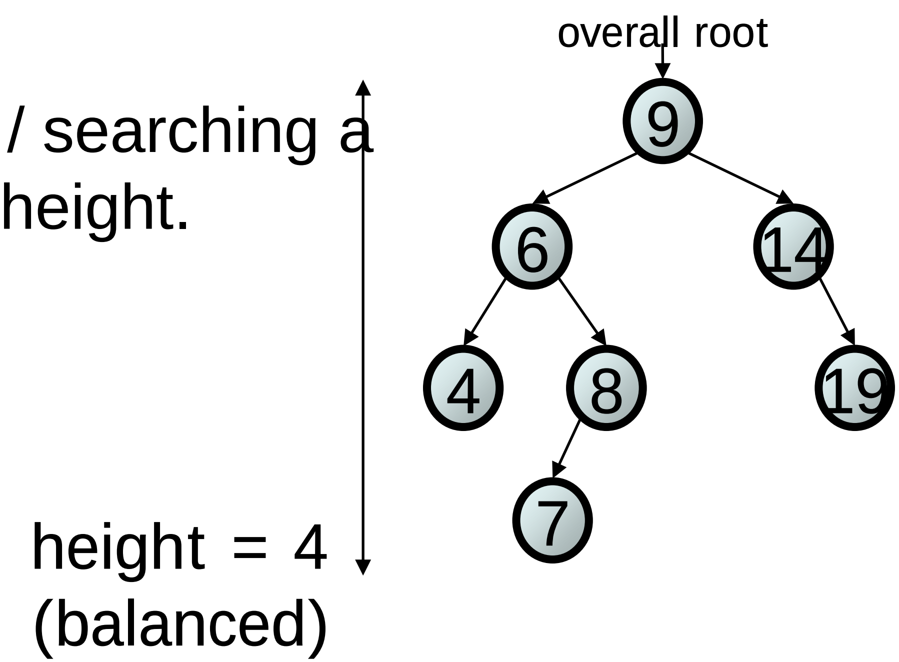
Searching BSTs

- The BSTs below contain the same elements.
  - What orders are "better" for searching?



Trees and balance

- **balanced tree**: One whose subtrees differ in height by at most 1 and are themselves balanced.
  - A balanced tree of N nodes has a height of ~ $\log_2$ N.
  - A very unbalanced tree can have a height close to N.
  - The runtime of adding to / searching a BST is closely related to height.



Tree rotation

- Tree rotation can rebalance trees when height of subtrees differs by 2 or more